Technical Manual

Ultrasonic Cat Softener

Group Cassia

MTRX 3700 Mechatronics 3  
Major Project 2009

**Note!!! Everything in red is a comment or placeholder, and MUST be deleted or replaced.**

**You should also delete sections that are not appropriate to your project.**

Date : 4 November 2009

Revision : 0.1

Prepared by : Group Cassia:

Distribution : MTRX 3700 Lecturer (David Rye)

# Introduction

## Document Identification

This document describes the design of something. This document is prepared by Group ZULU for assessment in MTRX 3700 in 2100.

## System Overview

Cassia talking scales is a portable microprocessor-controlled scale designed to be used by the average consumer with ease of use for the visually impaired. It can be modified by the client to display in either grams or ounces with a weight limitation of 1kg load (1000grams, 35.2739 ounces) at any given time.  
  
  
All user interface is driven by a keypad list interface. Options are listed and results displayed on an LCD screen as well as spoken by the scales through the integrated speaker.Document Overview

## Reference Documents

The present document is prepared on the basis of the following reference documents, and should be read in conjunction with them.

“PIC18F452 Data Sheet”  
“Winbond TTS Manual”  
“Altronics Keypad interpereter”  
"MM74C922 16 Key Encoder Datasheet "  
"ALTRONICS Z7000A LCD Module Datasheet"  
"Samsung KS0066U LCD Controller

### Datasheet Acronyms and Abbreviations

lists the acronyms and abbreviations used in this document.

Table -: Acronyms and Abbreviations.

|  |  |
| --- | --- |
| **Acronym** | **Meaning** |
| ACFR | Australian Centre for Field Robotics |
| CTS | Continuous *or* Continuously |
| PDL | Program Design Language |
| TBD | To Be Done, *also* To Be Defined |
| TLA | Three Letter Acronym |
| USYD | The University of Sydney |
| PIC | PIC18F452 |
| LCD | Liquid Crystal Display |
| MPU | Microprocessor Unit |
| A/D A2D | Analogue to Digital Converter |
| ASCII | American Standard Code for Information Interchange |

# System Description

This section is intended to give a general overview of the basis for the something system design, of its division into hardware and software modules, and of its development and implementation.

## Introduction

Give a technical description of the function of the *whole system*, in terms of its constituent parts, here termed *modules*. Generally, a module will have hardware and software parts.

### Voltage Measurement Module

The scales hardware is a cantilever beam approximately 30cm long and 6cm wide. When masses are added to a pan on the end of the beam, the deformation of the beam is measured by 4 strain gauges located near the fixed end of the beam. The resistance of these sensors is measured, and then normalised to a 0-V++ range by an op-amp.   
The voltage produced is measured by the A/D converter on the PIC (PORTE<1>), and recorded for further processing at a later time. The number saved is a 10-bit binary number, with 0 representing 0V, and 1024 representing V++ (negative voltages will be interpreted as 0V).  
V++ is generated by a variable voltage divider circuit. This can be manually adjusted using two trimpots (coarse and fine), to generate a voltage between 0 and 5V. By default this is set to 3.7V, allowing the full range of 0-1000g. If the scales hardware changed or needed to be adjusted, this enables an easy way to keep the system performing to specification.

### Calibration and Voltage to Weight Conversion

Based upon a linear gradient and intercept that is set during the calibration mode, the function getWeight() will look at the last voltage stored and convert it to a value of either grams or ounces (using a separate linear equation for the desired unit). Once converted, the N most recent values of mass are saved to a circular buffer, which is averaged and then converted into a string of characters to be displayed on Hyperterm or the LCD.  
  
Calibration mode allows a technical user to adjust the accuracy of the scales. After entering factory mode, the user may enter calibration mode. The scales will record the voltage generated by the scales when there is no mass on it, and also when some non-zero mass is placed on the scales. Using the serial receive capability via hyperterm, the user is prompted to enter the known mass on the scales, which is interpreted as a number. These two datapoints are passed to the calibrate() function, which will generate a new gradient and x-intercept that will be used for future weighings.

### String Storage and Retrieval

There is a very limited amount of available RAM on the PIC, so most of the strings that are used by the text to speech, LCD display, and the serial transmission functions are stored in the FLASH memory until they are ready to be used. When a function wants to send one of these strings, they access it via the function "get\_rom\_string(unsigned rom char\*)", which will fetch the sequence of characters via the TLATCH register, and then copy the string to a static location in local memory where other parts of the program can send it as if it were simply available in RAM. Since this location is shared by many functions, these values are copied to one of the output streams as soon as it is possible, because there is no guarantee that this temporay string will not have changed.

## Operational Scenarios

Describe how the system is to be used. There may be several different ways that it can be used, perhaps involving different users, or classes of user. Present use case diagrams here if you are using them. Each operational scenario is a part through a use case diagram – a way of using the system, with different outcomes or methods of use.

You should also consider the various failures that may occur, and the consequences of these failures.

## System Requirements

The operational scenarios considered place certain requirements on the whole something system, and on the modules that comprise it.

Statement of requirements that affect the system as a whole, and are not restricted to only a subset of its modules.

## Module Design

## Module Requirements: Module X

### The A/D Converter system is a core part of the system, as it is used to obtain readings from the scales hardware.

#### Functional Requirements

The functional requirements of the A/D module are particularly concerned with taking samples at correct intervals. This is important to guarantee constancy and reliability of the output.

##### Inputs

The A/D system requires a 5 V supply voltage to operate. The input signal must be an analogue voltage on one pin between 0 and 5 V.

##### Process

The module must operate fairly rapidly, as its operation will be performed quite frequently. A low computational complexity is highly desired.

##### Outputs

The output of the A/D converter must be a 10 bit digital value, stored in a 16 bit integer. This value must be a consistent representation of the input voltage. An input voltage of 0 V should output 0, and an input voltage of 5 volts should output 1024, thus giving a full range of possible values.

##### Timing

The sampling rate must be strictly timed such that it is regular and predictable. A suitable frequency is to be chosen, and the A/D converted must take samples at a rate as close to that frequency as is feasibly possible.

#### Non-Functional Requirements

The technical nature of the A/D module means that there are the non-functional requirements are strictly secondary to functional requirements. However, during the design process several possible non-functional requirements were specified.

##### Performance

Whilst the performance of the real-time part of the module is a functional requirement, any other sections can, theoretically, take some time to execute. However, it would be preferred if computational complexity for these parts is at a feasible minimum.

##### Interfaces

Ideally the A/D system should be entirely modular. Unfortunately, the complicated nature of the system makes this unlikely; see below in Design Constraints.

##### Design Constraints

The design of the microcontroller is such that it makes it quite difficult to build the system in a completely modular fashion. In particular, the limit to only two distinct interrupts means some parts of the system that would be preferably separated cannot be. In addition, the complex real-time nature of the system makes it somewhat difficult to completely separate the A/D functionality, which is a core part of the program, from other sections.

### Calibration

The calibration is an important part of ensuring the accuracy of the entire system, and is required to be able to accurately adjust the way the program converts the raw voltage from the scales strain gauges into numerically represented mass.

#### Functional Requirements

The main function requirement of calibration was accuracy of results, since they would directly relate to the accuracy of the entire system.

##### Inputs

The calibration needs to receive voltages and masses as input, in order to create a conversion equation. Unfortunately, these inputs are prone to being highly inaccurate. The strain gauges supplying the voltage are greatly affected by vibrations and oscillations, and the input masses themselves can be inaccurate. Additionally, a mass entered by the user can be inaccurate, either because of their own error in weighing, or if not enough significant figures are entered (particularly a problem when dealing with ounces).

##### Process

The internal process of the calibration module has to be as accurate as possible. Accuracy to many significant figures is preferred over integers.

Ideally, the module’s calculations should be able to produce an accurate calibration, which will produce a weight value with a maximum of ±1 g error, and it should be able to do so for any mass from 1 g to 1 kg.

##### Outputs

The output of the calibration module should ideally be as accurate as possible. Floats are preferred over integers, because this will enhance the accuracy of other parts of the program. Additionally, it must be able to output in both ounces and grams, no matter the input unit.

#### Non-Functional Requirements

Since the calibration module is mostly concerned with supplying accurate data to other parts of the system, the non-functional requirements are less important than the functional. However, there are still several aspects of non-functional requirements, which were considered in some depth during design of the module.

##### Performance

The computational complexity of the calibration module was a minor consideration because it would only be performed occasionally, and with not in real-time.

##### Interfaces

Simplicity for the user was an important issue. Whilst it would be possible to get a quite accurate system by creating a table of all possible voltage and mass combinations up to 1 kg, this would be an extremely difficult system to use. Ideally, the system should be able to produce a decent calibration with just two distinct voltage and mass readings.

##### Design Constraints

The limitations of the C programming language and the PIC microcontroller meant that extremely complicated calibrations were out of the question. It is difficult to perform complicated calculations, and too many calculations lead to a variety of errors, including rounding errors. The best solution to the calibration problem would likely be an n-degree polynomial line of best fit, however doing this using the tools available is quite impractical.

### Serial Transmit

#### Functional Requirements

This section describes the functional requirements of the Serial Transmit module – those requirements that must be met if the module (and system) is to function correctly.

##### Inputs

The input into the serial transmit module is encoded as a string of ASCII characters terminated by a null character (‘\0’, hex value 0x00). These strings may be dynamically generated by the program, as with numerical mass values, or defined prior to run-time, as with menu displays.

There is no correction or sanitisation of input. The module will transmit whatever string is passed to it.

The requirements for correct operation are:

* that no single string is longer than the size of the transmit buffer, which is 64 characters long;
* that many shorter strings are not transmitted immediately after each other, thus exceeding the size of the buffer with the cumulative length of all of the strings added together.

##### Process

The module must copy the input string into a circular buffer, ending when the null-termination character is reached. It must then copy one character at a time out of the buffer into the transmit shift register.

##### Outputs

The serial transmit module outputs ASCII characters over an RS232 serial connection, into the HyperTerminal terminal emulator via a DB9 male-to-male straight-through cable.

##### Timing

Timing is specified by the RS232 protocol. The connection parameters are as follows:

|  |  |
| --- | --- |
| Baud Rate | 9600 |
| Message length | 8 |
| Parity | None |
| Stop bits | 1 |

##### Failure modes

No specification was made about maintaining functionality during hardware failure. It is worth noting that the system is dual-input, dual-output, so output can be provided via the LCD module if serial transmit becomes non-functional.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

It is necessary to consider timing in sending a string to the transmit module. The amount of time required between transmit operations obviously depends on the string length. An enforced delay of 10,000 clock cycles is provided in the module. Further delays may be necessary if transmitting multiple strings of moderate size (greater than about half the buffer length). If sufficient delays are not provided, basic functionality will be retained, but the strings may be badly formed with the loss of some characters due to overwriting. Testing with the planned strings is the best method to ensure correct functionality.

##### Interfaces

It is desirable for the serial transmit module to be able to be passed a simple null-terminated string, as this is the common format used for data storage in other parts of the program..

##### Design Constraints

The processor clock speed of 4 MHz is relatively low, meaning that the module cannot afford to poll a bit to determine when each character has been sent, and must use the transmit interrupt instead. RAM is also severely constrained, restricting the length of the transmit buffer. The software part of the module must be programmed in the ANSI C language.

### Serial Receive

#### Functional Requirements

This section describes the functional requirements of Serial Reception – those requirements that must be met if the module is to function correctly.

##### Inputs

Inputs into the serial reception module are ASCII characters transmitted over an RS232 serial connection. These inputs are made in the hyperterminal terminal emulator via a DB9 male to male straight through cable.

##### Process

The characters received are stored in a circular buffer and are progressively parsed in the program’s main loop.

##### Outputs

The output produced is a command character which is parsed to change the system’s settings.

##### Timing

Timing on the serial receive module is specified by the RS232 protocol. The connection parameters are as follows:

|  |  |
| --- | --- |
| Baud Rate | 9600 |
| Message length | 8 |
| Parity | None |
| Stop bits | 1 |

##### Failure modes

In case of failure, inputs are still possible on the local input device – keypad.

#### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

##### Performance

Reception of a character via the serial protocol should launch the system into USER\_REMOTE mode.

##### Interfaces

The user interface provides updating information on what characters have what effect in the current mode.

##### Design Constraints

So as to only deal with inputs when they become available, the Serial Reception modules operates using the RC interrupt. The received character buffer is intentionally controlled at a low volume to conserve RAM.

### Functional Requirements

This section describes the functional requirements of Module X – those requirements that must be met if the module (and system) is to function correctly.

#### Inputs

Describe each external input, including signal encoding and timing, message encoding and timing, protocols, file formats, protection against input errors, etc, as relevant.

#### Process

Describe the internal signal transformations and/or computer processing functionality required within the module, required performance limits, and error tolerances as appropriate.

#### Outputs

Describe outputs that must be produced for the module to function correctly, including timing, frequency, protocols, etc as relevant.

#### Timing

Any required timing or latency specifications that must be met.

#### Failure modes

Required functionality (if any) in the event of failure of various nominated components.

### Non-Functional (Quality of Service) Requirements

Non-functional requirements do not need to be met for the device to have basic function, but are required to provide specific levels of performance or engineering quality.

#### Performance

Requirements such as computational loop time, accuracy, etc.

#### Interfaces

Qualities that are desirable in input and output interfaces, but are not required for functionality.

#### Design Constraints

Practical or commercial considerations, such as programming languages, processor or other hardware, etc.

## Conceptual Design: Module X

### Analogue to Digital Converter

Designing the basic operation of the A/D module was trivial: the A/D module is initialized, the program starts the conversion, and then the program reads the result. However, the actual operation of the module was found to be more complex.

The functional requirements for the module specify real-time operation with regular sampling at a known frequency. The original design used a timer and a compare function to trigger an interrupt, which was then used to perform the A/D conversion.

However, it was quickly realized that this did not fulfill all the functional requirements. Performing the A/D conversion inside an interrupt was slow and clumsy, and there was no way of guaranteeing that the actual conversion would perform and sample at the correct rate. Thus, another solution was researched.

It was found that the CCP2 module in the PIC microcontroller can be used with a Special Event Trigger to automatically begin the A/D conversion on a timer compare. In conjunction with a Conversion Complete interrupt, this enabled the program to perform the time consuming conversion outside the interrupt, and sample at precisely the desired rate. This solved most of the previous problems.

In addition to this, in order to improve the speed of operation of the interrupt, it was decided to modify the system so the interrupt did not attempt to convert the value in the A/D register into an integer; it merely stored it for later conversion. By setting a flag indicating a new A/D value, it was possible to move this conversion into the mainline of the program, thus saving on computational complexity in the interrupt.

Shown below is a representation of how the two result registers, the Analogue to Digital Result High (ADRESH) and the Analogue to Digital Result Low (ADRESL), correspond bitwise with the integer result.

ADRESH

H1

0

0

0

0

0

0

L2

L3

L6

L5

L4

L7

L0

H0

ADRESL

L1

Integer result

L2

L3

L6

L5

L4

L7

L0

L1

H1

0

0

0

0

0

0

H0

#### Assumptions Made

It was always assumed that the input was correct; that the strain gauges on the scales were working correctly. See Constraints, below.

#### Constraints on A/D Converter Performance

The accuracy of the A/D converter module is highly dependent on the accuracy of the scales. If the strain gauges were incorrect, or there was some sort of hardware problem, then the A/D converter would produce incorrect results.

### Calibration

Early tests with the hardware indicated that the relationship between input voltage and mass was fairly linear. An oscilloscope was used to obtain voltages for a variety of test masses. Whilst the complexity of the beam meant that true linearity was unlikely, it was sufficiently so that it was deemed acceptable to use linear fit for calibration.

To improve accuracy and speed of calculation, it was decided that the calibration and weigh methods should go directly from the raw A/D value to the mass, rather than converting the voltage to a value between 0 and 5 V first. Thus, the following graph was produced using the same data as above, but inverting the axes and using voltages as if they were the 10 bit A/D values which the microcontroller would see.

Initial prototypes of the calibration module had a very simple method, based on the above research. Using integer arithmetic for speed, the module created a linear line of best fit using a simple equation, involving calculating gradient and y‑intercept from the two data points. However, this was quickly found to be highly inaccurate for the following reasons:

* Integer arithmetic does not provide sufficient accuracy and is prone to rounding errors
* The simple equation did not fully utilize the potential accuracy available
* Two data points were a poor representation of the entire voltage-mass relationship

The last point was deemed too difficult to fix, as a more complicated fit line would be both expensive both in computational and memory terms, and it would be highly susceptible to errors in the microcontroller C environment.

After integration of some of the modules of the system, it was found that computational complexity was far less important than originally thought. Thus, it was simple to change the calibration module to use float arithmetic.

Whilst the basic method of using a straight linear fit between two points was acceptable, it was possible to improve the accuracy of the results. By multiplying the 10 bit A/D voltage values by 64, they took up a full 16 bit integer. Whilst not useful in itself, this meant that when a (fractional) gradient is applied, a larger number remains, which is more accurate. The basic weigh function was modified to:

The gradient was used as an inverse because the value would always fall in the range (0,1). By calculating and storing the inverse a greater degree of accuracy could be achieved.

Assuming two data points, at 0 mass with some voltage, and another known mass with some other voltage, the gradient-1 and offset variables can be isolated.

It is now quite easy to use these equations to calculate *g* and *f*.  
  
Another consideration was the need to produce the results in both ounces and grams. It was decided that it was impractical to perform the whole process with different mass units, and thus the most effective method would be to convert the result. It was decided that a unit conversion function would be used. To convert the g value, it would need to be divided by the conversion factor, whilst the *f* value would need to be multiplied.

Once all of these problems were solved, it was simple to implement the module in software.

#### Assumptions Made

It is assumed that all the values used during calibration are roughly accurate. Specifically, it is assume that the known mass is entered properly. An incorrect input will likely produce an extremely incorrect output.

#### Constraints on Performance

The inaccuracy of the scales is a serious problem for the calibration. Even slight fluctuations in the voltage caused by small oscillations can throw off the result significantly. If a mass smaller than the calibration mass is weighed, it is likely to be more accurate that if a larger mass is weighed, because the interpolation is more accurate than extrapolation.

### Variance calculation

This calculates the variance inherent in the calculated mass, based on of the buffer of voltage values. This buffer forms a ‘population’ of finite size, pre-defined as VBUFSIZE = 8.

Thus the general formula for population variance is applied:

for number of samples , individual samples and mean . Here VBUFSIZE, is each voltage sample in the buffer, and is the mean voltage. This mean voltage is already calculated by a separate function so it can be easily accessed by the variance function.

The decision was made to take the variance of the voltage buffer and then convert it to a mass value, rather than taking the variance of several calculated mass values, because our setup removes the need to store a separate buffer of mass values.

Floating point arithmetic is used to provide an accurate value of the deviation, which is usually much smaller than the actual mass value. The voltage values are stored as unsigned integers, but are cast to floats for the mathematical operations to avoid truncation, then back to an unsigned integer for reporting at the end.

The variance in voltage is then converted into a variance in mass, for reporting to the user, by using the same calibrated mass-voltage conversion as for the weight. However, there is an important difference. Because the variance deals only with differences in voltage rather than absolute values, there is no need to apply an offset; only the gradient in the mass-voltage line is necessary. As an example, consider a perfectly stable voltage which thus has a variance voltage of 0. Passing this into the voltage-to-mass calculation will result in , giving a non-zero variance mass when the answer should be zero. Instead, for variance calculations the correct formula is only. This is implemented with a flag, as explained in the ‘Software design’ section.

### Serial Transmit

Serial Transmit is implemented using the PIC’s USART communications functionality. This must be correctly configured in software, a process which is described elsewhere in this manual.

One software function takes the string passed to the module and copies it into the transmit buffer. This buffer is circular and uses an incrementing insertion pointer that is reset to zero when it reaches the end of the buffer. It copies each character from the provided string until it reaches the null-termination character, which is not copied.

A second software function takes a character out of the buffer and places it into the transmit shift register, where it is automatically transmitted out by the PIC. Completing this operation triggers the transmit interrupt, indicating the system is ready for another character, and this function is called again to fetch the next one. It uses an incrementing removal pointer to index the transmit buffer, which is also reset to zero when it reaches the end of the buffer.

Transmission continues until the removal and insertion index pointers have the same value, indicating all of the string has been transmitted.

The data flow is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| String in memory:  **“Hello”** | String copied into transmit buffer   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘\0’ | | In interrupt, characters copied from transmit buffer into transmit shift register:   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | ‘H’ | ‘e’ | ‘l’ | ‘l’ | ‘o’ | ‘\0’ | | ↓ | | | | | | | ‘H’ | | | | | | | Character transmitted from transmit shift register via USART to HyperTerminal. |

**Rationale**

Using the inbuilt USART functionality was the simplest method to implement. Having a transmit module that can be directly passed a string, rather than having to write one character at a time into the transmit shift register, greatly simplifies ease of coding elsewhere in the program.

**Constraints on Serial Transmit Performance**

As described above, the timing issue is the greatest constraint. Hardware errors such as inoperative cables or connections will prevent correct operation, as will incorrect setup of HyperTerminal not in accordance with the settings provided above.

### : Serial Recieve.

Serial Receive is implemented using the PIC 14F852 microcontroller’s inbuilt USART communications functionality. The dataflow is as follows:

Control Character

Hyper-terminal

RS232

USART

Input buffer

Port C pins 6 and 7 are used for serial input and output respectively.

To communicate with Hyperterminal, the microcontroller’s USART module needs to be configured for communication. RC interrupts are enabled and set for high priority.

|  |  |
| --- | --- |
| TRISCbits.TRISC6 = 0; | PortC6 is input |
| PIR1bits.RCIF = 0;  PIE1bits.RCIE = 1;  IPR1bits.RCIP = 1; | Clear RC flag.  Enable RC interrupt.  High Priority for RC interrupts |
| RCSTAbits.CREN = 1; | Enable RC |
| INTCONbits.PEIE = 1; | Enable Peripheral Interrupts |

Increment insertion pointer around buffer.

On Serial RC interrupt

Copy Serial Register to insertion pointer position

Reset flags for interrupt.

Serial RC complete

Operation of the Serial recieve module is described simply in the diagram below.

#### Constraints on Serial Receive Performance

Serial Receive is quite robust and is able to handle non operative characters.

Now, ***for each module***, give the outline of how it will work. In this section it is appropriate to present

* The rationale for the design decisions that were made – *why* things   
  were designed the way they were
* block diagrams,
* mathematical models and algorithms,
* data flow diagrams
* state-transition diagrams
* listings of input and output formats
* listings of message and data formats
* responses to identifiable error conditions
* responses to identifiable failure conditions

as appropriate for each module.

### Assumptions Made

State any assumptions made.

### Constraints on Module X Performance

State any constraints that may prevent the design from satisfying its requirements.

# User Interface Design

Give a detailed description of the design of the user interface. This will give the reader a good view of how the system functions from the user’s perspective.

## Classes of User

If there are different user interfaces presented to different classes of users, define these user classes, and how access by the various user classes is enabled or disabled.

## Interface Design: User Class Y

### User Inputs and Outputs

Description of how the user presents inputs to the system, and how the system responds to those inputs. Include a description of how the user knows the state of the system.

### Input Validation and Error Trapping

Describe how the system validates user input, and how operator errors are trapped and can be recovered from.

# Hardware Design

Give a detailed description of the design of hardware. The description should include mechanical drawings, location diagrams, electrical circuit schematics, circuit simulation or test results, PCB overlays, wiring diagrams, connector pin-out lists, pneumatic/hydraulic circuit diagrams,

## Scope of the Something System Hardware

Statement of what is, and what is not, being designed and described here.

## Hardware Design

### Power Supply

Power supply method and rating, fusing, distribution, grounding and protective earth as appropriate.

### Computer Design

Description of computer hardware, including all interface circuitry to sensors, actuators, and I/O hardware.

### Sensor Hardware

### Actuator Hardware

As appropriate.

### Operator Input Hardware

### Operator Output Hardware

### Hardware Quality Assurance

Describe any measures that were taken to control (improve) hardware quality and reliability – Heartbeats, brownout conditioning/resets, reset conditions, testing and validation, etc.

## Hardware Validation

Details of any systematic testing to ensure that the hardware actually functions as intended.

## Hardware Calibration Procedures

Procedures for calibration required in the factory, or in the field.

## Hardware Maintenance and Adjustment

Routine adjustment and maintenance procedures.

# Software Design

The software requirements and overview have been dealt with elsewhere in this document. The present section addresses the design and implementation of the software that forms the X system.

## Software Design Process

How you went about designing the software – top down, bottom up, OOD, functional view, etc.

### Software Development Environment

The tools that were used, both software (compilers, assemblers, etc) and hardware (development boards, etc.).

### Software Implementation Stages and Test Plans

Describe the way you went about implementing the software - staged implementation, pseudocode (PDL), unit testing procedures, integration testing, etc. Identify dependencies – e.g this had to be done before that, etc.

## Software Quality Assurance

Describe any measures that were taken to control (improve) the software quality – code or documentation standards, code walkthroughs, testing and validation, etc.

## Software Design Description

### Architecture

Describe the high-level architecture of the software – that is, the top-level flow of control, and how the various functional modules communicate.

In this section, you can put state transition diagrams, sequence diagrams, etc.

### Software Interface

Describe the public interface of each software module.

### Software Components

#### Analogue to Digital Converter

##### Interface

After power up, the A/D converter hardware must be initialized. This can be performed using the initialise\_AD\_converter method in the Setup module. Once this is performed, and the global interrupts enabled, the A/D converter should automatically start sampling at a frequency of 10 Hz. To get the A/D result, the newADValue flag should be checked. If it is set, then it should be cleared, and the A/D voltage result can be obtained by calling the getVoltage function.

##### Components

There are three important functions in the A/D converter module: the initialization, the interrupt, and the getVoltage method.

##### initialise\_AD\_converter

This method primarily sets all the registers required to make the A/D convert module work. It starts by enabling the A/D module, and setting it to Fosc/8 timing. Then, pin A1 is chosen as the input pin. This was used rather than pin A0 because on the PIC evaluation board a potentiometer is connected to pin A0. This pin is also configured to input. The A/D converter is set to 1 analog channel with VDD and VSS references. The output registers are set to be right justified, such that it is easier to extract and use the whole 10 bit result value.

Timer 3 is enabled and set to use a x8 prescaler. This is important for the operation of the CCP2 Special Event Trigger which starts the A/D conversion. The CCP2 module itself is initialized, and set to the Special Event Trigger mode. The CCP2 compare register is set to a precalculated value to produce trigger ever 100 ms. The A/D Conversion Complete is enabled, and set to low priority.

Timer 0 is briefly enabled, and the program waits for it to time out before continuing. This is to allow the A/D converter time to start up before any conversions are attempted. Timer 0 is then disabled, and reset.

After execution of this method, the A/D converter should be fully functional. When the global interrupt flag is enabled, the A/D converter will automatically begin sampling every 100 ms, using the low priority interrupt.

##### low\_ISR

The interrupt service routine for the low interrupt is used by several modules. This section only covers its use by the A/D conversion module.

The interrupt checks whether the A/D Conversion Complete interrupt is set, and if it has been triggered. If so, it performs the actions stated below, and then resets the flag.

The A/D result registers, ADRESH and ADRESL, are saved to global variables. This fundamentally takes a snapshot of the current value of the A/D result. The newADValue flag is then set. This allows the mainline of the program to detect the new value, and extract it from the stored values using getVoltage.

##### getVoltage

This function is designed to return a 16 bit integer containing the 10 bit voltage value extracted from the A/D converter, as specified in the functional requirements.

At a basic level, it was designed to take the high character, place it into an integer, and left shift it 8 places such that it occupied the high byte. The low character was then to be added. Unfortunately, the technical limitations of the PIC microcontroller and C compiler made this more difficult.

The first problem was that the bit shifting in the C compiler did not comply with normal ANSI C standards: instead of performing bit shifting, it seemed to perform a rotation. Rather than try to correct this problem, it was deemed simpler to replace the bit shifting by a multiplication: in this case, a multiplication by 256.

The second problem was that converting from a character to an integer caused the high bits in the integer to be undefined. This was particularly problematic when attempting to add the low character in to the integer result. A workaround was found, involving copying the character into a temporary integer first, using a bitwise AND operation to clear the top 8 bits, and only then adding it to the result.

Thus, using the above design, the function adds the high character to the integer result, and multiplies it by 256. The low character is then converted to an integer, the high byte masked out, and then it is added to the result. Finally, the result is returned.

### Calibration

#### Interface The calibration methods are part of the Scales module, but are accessed primarily by an external module containing the user interface, in this case the States module. The calibration can be performed automatically by calling the calibrate function. The following values are required as parameters:

* The A/D voltage when the scales are empty
* The A/D voltage when the scales contain a known mass
* The numerical mass correlating to the previous voltage
* The units that the mass is in

The voltages should be obtained using the getVoltage method in the Scales module. The numerical mass is input by the user, and the units should be specified.

#### Components

There are two functions in the calibration module: calibration, and convert\_unit.

#### calibration

The calibration method performs the actual calibration. It requires, as parameters, the four values specified above: unsigned integers with the voltages for 0 and some other mass, a signed integer for the actual mass, and a character of 0 for grams and 1 for ounces.  
  
The method starts by calculating the difference in voltage between the upper and lower masses. If this value is 0, the function quits, as this value would result in a divide-by-zero error.  
  
The function then proceeds to calculate the calibration parameters as specified in the Conceptual Design: Calibration section. Specifically, the following equations are used:

The second equation was, for technical reasons, split into two parts: the first calculating the value, and the second applying the negative sign.

Finally, the results are converted to the non-active unit (gram -> ounces or ounces -> grams) and all the results are stored in memory.

#### convert\_unit

The convert\_unit function is simply designed to convert a mass from grams to ounces, or ounces to grams. It uses a lookup table of conversion factors to multiply the input mass. Float arithmetic is used to assure maximum accuracy.

## Preconditions for Software

### Preconditions for System Startup

Describe any preconditions that must be satisfied before the system can be started.

### Preconditions for System Shutdown

Describe any preconditions that must be satisfied before the system can be stopped.

# System Performance

### Performance Testing

Give the results of testing conducted to determine the characteristics and performance of the system - memory usage, loop time, system accuracy, repeatability, ease of use, etc.

### State of the System as Delivered

A statement of your group’s opinion of the conformance of the system with the specification.

### Future Improvements

Present a prioritised list of improvements to be made in future releases, giving reasons for the improvement and priority rank.

# Safety Implications

By law (*NSW Occupational Health and Safety Act 2000*; *NSW Occupational Health and Safety*

*Regulation 2001*) all employees who design plant, machinery or equipment must identify foreseeable safety hazards associated with the equipment, and then assess and control the identified risks. Although this law does not apply directly to student designs, you should consider it here.

# Conclusions

**Appendix A: Title**

**A.1. Subtitle**

Appendices are used for any material that is not included in the main part of the report, usually because it would be distracting to the reader.

You would normally include **manufacturer’s data sheets**, but please do NOT include them in this report.

**Supporting calculations** should also go in an appendix.

If you use **Doxygen** (www.doxygen.org) for code documentation, generate rich text format (RTF) output, open the RTF in Word, copy, and paste it into an appendix.

Please do place your **code listings** in the following 2-column section. Make sure to use a non-proportional (mono-spaced, or “typewriter”) font, such as Courier New or Andale Mono.

Place code listings here, using style “Code Appendix” The column width will fit approximately 88 characters per column.

Each file should start on a new page, or at least at the top of a new column.